Vol.2018-SLDM-185 No.31
Vol.2018-EMB-49 No.31
2018/12/6

R2OS：WSNにおける共有資源管理と実行モデルの提案

井上 明紀[1,a)]　福田 浩章[1,b)]

概要：近年，Internet of Things(IoT) の発展により，Wireless Sensor Networks(WSN) が再び注目され，多くの企業が開発・製造・管理を行い活発化し多種多様な目的で利用されている．WSN は最小限のハードウェアモジュールで構成されており，安価で大量のデバイスを利用することができる一方，通常のコンピュータと比較すると性能に大幅な制約があり，ハードウェアリソースの利用を厳密に管理する必要がある．WSN で動作するアプリケーション開発者は，本来のビジネスロジックとは直接関係のないリソースの問題を考慮する必要があり，開発コストを高めている．そのため，ハードウェアの抽象化やスケジューリングなどを代理する WSN に特化した軽量 OS が提案されている．しかし既存の OS では，複数のプログラムが動作する複雑なアプリケーションは実行時にハードウェアリソースを共有する機能は備えておらず，それらを管理することは困難である．そこで本研究では，WSN に特化した新しいアプリケーションの実行モデルを定義し，それに基づいた OS の提案と実装を行う．提案する実行モデルでは，アプリケーションはハードウェアリソース間を繋ぐパイプラインとして定義し，特定の入力によって実行される．OS は内部で複数のハードウェアリソース管理やパイプラインの実行，及びスケジューリングを行う．その結果アプリケーション開発者はビジネスロジックの開発だけに集中することができる．

キーワード：WSN，OS

R2OS : Shared Resource Management Model for WSN

Akinori Inoue[1,a)]　Hiroaki Fukuda[1,b)]

Abstract: Wireless Senwor Networks (WSN) have been becoming hot topics because of the trend of Internet of Things (IoT). A number of companies are developing and managing sensor devices and using them for a variety of purposes. We can use a number of such devices because they are enough cheap, however each device prepares a limited hardware resources compared to traditional computers. Thereby developers should have additional concerns about the limited resources besides the essential application logic, resulting in increasing development cost. Although we can use lightweight operation systems (OS) that provide useful functions to manage such resource, they do not provide functions to share hardware resources among application programs running on the OS.
In this paper we propose a novel execution model for WSN and give a prototype implementation of the OS that follows the execution model. In the proposed execution model, an application is defined as a pipeline that connects between resources, and executed by the specific trigger such as receiving data from the network. Our OS manages hardware resources, schedules applications. As a result developers are able to concentrate on the core of the application logic.

Keywords: WSN，OS

1. 背景・目的

WSN は，多数のセンサノードを組み合わせてネットワークを形成し，ノードに取り付けられたセンサモジュールか

[1] 情報処理学会 IPSJ，Chiyoda，Tokyo 101–0062，Japan
[†1] 現在，芝浦工業大学
Presently with Shibaura Institute of Technology
[a)] ma17016@shibaura-it.ac.jp
[b)] hiroaki@shibaura-it.ac.jp

ⓒ 2018 Information Processing Society of Japan

Vol.2018-SLDM-185 No.31
Vol.2018-EMB-49 No.31
2018/12/6

らデータを収集する無線ネットワークである．近年の IoT の発展や組み込み向けのハードウェア性能の向上 [1], [2] から，WSN の利用も見直され，データを集計する単純なものから複雑な要件へ利用されている．センサノードデバイスはその目的から大量に投じることを想定し安価なモジュールで構成されている．そのため，一般的なコンピュータに比べ CPU，RAM，Flash メモリ，バッテリといった様々なハードウェアリソースの制限が存在するのに加え，要件ごとの独立したハードウェアの構成と調整が必要である．これによりアプリケーション開発者は，センサノードで実行されるアプリケーションロジックとは異なるハードウェアの制御を考慮しなければならず，開発コストを高める要因となっている．そこで既存研究では，WSN に特化した組み込み向け OS を提案している [2]．WSN の OS は，通常の OS よりも容量や実行時のオーバーヘッドを削減し，ハードウェアリソースの抽象化と長期的実行のためのスケジューリング機能を提供している．アプリケーション開発者はこれらを利用することで，抽象化されたハードウェアリソースを容易にかつ厳格に管理することができる．スケジューリングに関しては，提案されているほとんどの WSN の OS ではオーバーヘッドの少ないイベント駆動型実行モデル [3] を採用しているが，近年では複数のロジックを持つアプリケーションが同時に実行されることを想定して，プリエンプティブスケジューリング [4] を行うアーキテクチャやイベント駆動型実行モデルとのハイブリッドとして提供する OS[5] が提案されている．しかし複数の処理が実行される場合には，スレッドの切り替えや共有リソースのロックといったスレッド間の制御を意識しなければらない．そのため，複雑なロジックを要するアプリケーションの場合，様々なロジックが 1 つの共有リソースを利用するため，意図しない動作が発生しやすい．また意図しない動作が発生した場合は，不具合の再現が難しく原因の究明や解決が困難である．そして各ロジックが複雑な依存関係を持つことにより複数人での開発も困難となる．

そこで本論文では，アプリケーションを共有リソースのロックといった複雑な制御を必要とせず実行できる，R2OS を提案する．R2OS はアプリケーションをプロセス単位で実行し，メモリ空間の分離とラウンドロビンスケジューリング [6] による並列での複数の実行を保証する．その上で，R2OS はアプリケーションをリソースとリソースをつなぐ複数のパイプラインとして実行する．接続したリソースがハードウェアを管理するため，アプリケーション開発者は I/O などの制御を一切意識せず様々なアプリケーションの開発を行うことができる．

2. アプローチ

本節では，はじめに WSN のアプリケーション開発における問題を実際の開発を想定して明らかにする．次にこの問題を解決する R2OS のアプローチを述べ，最後に R2OS を利用したアプリケーションの実現方法を述べる．

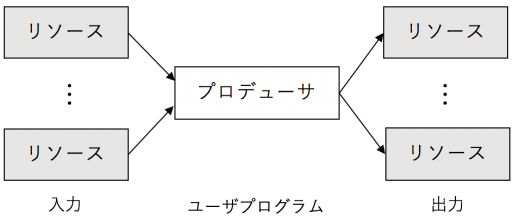

図 1 プロデューサとリソースの関係

2.1 アプリケーション開発の問題

本節では，一般的な WSN のアプリケーション [7] の 1 つである継続的なデータ検出を例にアプリケーション実装時の問題を述べる．このアプリケーションは複数のセンサノードからサーバへデータを定期的に集計する．WSN では，一般的にデータを検出するノードと，それらデータを集約しサーバに転送するシンクノード [8] が存在するが，ここでは，ノードで動作するアプリケーション開発を考える．WSN での通信距離は限られるため，一般にマルチホップすることでシンクノードと通信する．そのため，各ノードはシンクノードまでの経路を確立する必要がある．その後，各ノードはセンサーなどを利用して取得・加工したデータ，あるいは隣接ノードから転送されるデータを経路情報をもとに転送する．これを動作を繰り返すことで，シンクノードにデータを集約する．この動作をそのまま実装すると，次のようになる．

(1) 経路情報を生成しキャッシュする．
(2) センサを利用してデータを検出し，キャッシュされた経路情報を利用してデータを送信する．
(3) 隣接ノードからデータを受信し，経路情報を利用して転送する．

この方法には次のような問題が存在する．単一スレッドで経路情報の生成を開始すると，その他の処理がブロックされてしまう．経路情報の生成には複数のノードとの通信を行う必要があるため，ブロックされる時間が長くなり，その間センサを利用したデータの検出ができない．既存の WSN の OS では, プリエンプティブスケジューリングによるスレッドの並列実行でこの問題を解決している．一方，マルチスレッドで並列実行する場合，共有リソースをアクセスするときにはロックなどの機構を用いて同期をとる必要がある．例えば経路情報を複数のスレッドで共有する場合には，同時書き込みを防いだり，読み込み中の書き込みを防ぐような処理を施す必要がある．

2.2 R2OS のコンセプト

2.1 節で述べたように，複数スレッドによる並列処理を行う場合，共有リソース (メモリやネットワーク) の処理が

ⓒ 2018 Information Processing Society of Japan

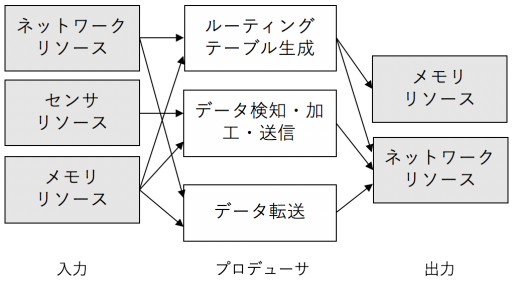

図 2 R2OS で実行されるデータ検出システムのアプリケーション例

問題となる．一般的な WSN 用の OS では，排他制御の仕組みは提供しているが，共有リソースの扱いは開発者に依存している．つまり，開発者が正しく共有リソースを保護しない限り，共有リソースの安全は保証されない．そこで R2OS では，アプリケーションを 1 つ以上の入力を受け取り，1 つ以上の出力を行うプログラム (以下，プロデューサと呼ぶ) と定義し，図 1 に示すようにリソースを R2OS が管理することで排他制御の問題を解決する．プロデューサは，1 つ以上の入力を待機し，必要な入力が全て揃うと R2OS によって起動される．また，入力に何らかの加工をしたあと，1 つ以上の出力に行って終了する．そして，入出力に関する処理を R2OS が安全に行うことで，アプリケーションの開発者から複雑な排他制御の仕組みを隠蔽する．この結果，プロデューサ単体は副作用を持たないプログラムとなるため，開発効率やデバッグも容易になる．また，入力と出力を繰り返しながら動作するアプリケーションを記述する場合には，複数のプロデューサを定義し，入出力を契機に連動して動作させることで実現する．

2.3 R2OS でのデータ検出アプリケーション

2.2 節で述べたように，R2OS のプロデューサは副作用のないユーザプログラムであるため, プログラムを実装する上で自由度が下がるが, 様々な入出力リソースとプロデューサを連動することにより柔軟にアプリケーションを実現することができる．例えば，2.1 節で述べたデータ検出の場合，以下の 3 つのプロデューサでアプリケーションを構成できる．

- P1: 経路情報生成
- P2: データ検知・加工・送信
- P3: データの受信と転送

また，これらプロデューサの入う出力となるリソースは以下の 3 つである．

ネットワークリソース ノードからのデータ送受信を抽象化したリソースである．ネットワークリソースへの出力は，アドレスを含める必要がある．

センサリソース タイマ割り込みによって一定のタイミングで測定したデータを表すリソースである．

メモリリソース 永続化するデータを保持するためのリソースである．データ検出アプリケーションでは経路情報の保持に利用する．

これらリソースと 3 つのプロデューサの関係を図 2 に示す．これらプロデューサとリソースで，このようにデータ検出アプリケーションを実現するか概説する．

P1:経路情報生成 このアプリケーションでの経路情報とは，シンクノードへのホップ数と，隣接ノードのペアである．隣接ノードからネットワークを介してデータを受信すると，ネットワークリソースを入力とする P1 が起動される．P1 は受信したデータが経路情報だった場合，経路をメモリリソースに書き込み，ホップ数を加算してネットワークリソースに書き込むことで隣接ノードに経路を配布する．

P2:データ検知・加工・送信 センサリソースがセンサを利用してデータを検出すると，センサリソースを入力とする P2 が起動される．P2 は検出したデータを必要に応じて加工し，メモリリソースに保存された経路情報を参照することで，ネットワークリソースに検出データを書き込む．

P3:データ転送 P1 同様，隣接ノードからネットワークを介してデータを受信すると，ネットワークリソースを入力とする P3 が起動される．P3 は受信データの種類がデータ転送だった場合，メモリリソースの経路情報を参照しネットワークリソースに書き込むことで，マルチホップを実現する．

このように，R2OS を利用する場合，各プロデューサの開発者は，入力するリソースと出力するリソースだけ意識すれば良く，各プロデューサが入出力を契機として連動して動作することで，目的のアプリケーションを実現できる．また，リソースの入出力は必ず R2OS を介して行うことで，開発者は排他制御を意識する必要はない．なお，各プロデューサが並列に実行される場合，R2OS は各プロデューサをラウンドロビンスケジューリングによって均等に実行することで，スタベーションを防止する．

3. アーキテクチャ

本節では，R2OS のアーキテクチャについて説明する．

3.1 排他制御

R2OS は, アプリケーション側に意識させず排他制御を実現するために, 共有リソースの入出力をアプリケーションに代わって行う．この節では, 排他制御が必要な場合と R2OS がどの様に排他制御を実現しているかを説明する．まず排他制御が必要性なケースは, 次で示す 3 つの入出力のタイミングで存在する．

- スレッドが共有リソースへの入力中に, 別のスレッドが出力することで入力のデータの不整合が発生する．
- スレッドが共有リソースへの出力中に, 別のスレッドが出力することで出力後のデータの不整合が発生する．

ⓒ 2018 Information Processing Society of Japan

- スレッドが共有リソースへの出力中に, 別のスレッドが入力することで入力データの不整合が発生する.

この問題は次の 2 つの入出力の制約によって解決できる.

- 共有リソースへの入力を開始した場合, 終了するまで他のスレッドの出力を禁止する.
- 共有リソースへの出力を開始した場合, 終了するまで他のスレッドの入出力を禁止する.

R2OS では他のスレッドの入出力を禁止するために, 割り込み禁止を利用している. 割り込みを禁止にすることによって, 現行のスレッドが直列で処理され他のスレッドの入出力処理が行われなくなる. これは並列で実行するスレッドはタイマ割り込みを契機とした, ラウンドロビンスケジュリングによって実行されるためである. スレッドの並列実行を実現するために, 実行中のスレッドはタイマ割り込みによって別のスレッドへ実行が移される. しかし, 割り込み禁止をすることでタイマ割り込みによる別のスレッドの切り替えが起きず, 現行のスレッドが割り込みを許可するまで処理される. これによって R2OS はを内部で実現している.

3.2 カーネル空間とユーザ空間

R2OS にはカーネル空間とユーザ空間が存在し，実行される全ての処理はいずれかの空間で行われる．3.1 節で述べた割り込み禁止する実行領域がカーネル空間である．ユーザ空間では実行される処理はスケジューラによって管理される．つまりユーザ空間上のプログラムは並列で実行される．ユーザ空間のプログラムがカーネル空間の処理とのやり取りを行うためには, システムコール API を通じる必要がある.

3.3 コンポーネント

各コンポーネントは，それぞれ役割を持ったプログラムである．この節では各コンポーネントについて説明する．

3.3.1 ドライバ

ハードウェアを抽象化したリソースの処理は，ドライバによって実装されている．ドライバは割り込みによる入力から，リソースマネージャへ出力を行う．ドライバは入出力を持つプロデューサと似ているが，ステートフルなプログラムでありカーネル空間上で実行される点で異なる．このことからドライバは, 様々なリソースを抽象化できる．例えば，シリアルポートからの受信をネットワークリソースの入力としてドライバが実装する場合は，シリアルポートから 1 バイトずつ割り込みでのデータを受信し，バッファリングを行い，パケットデータとして読み取った後にリソースの入力として生成する．これにより，ネットワークリソースの入力抽象化を行うことができ，プロデューサは抽象化された入力からネットワーク情報を容易に扱うことができる.

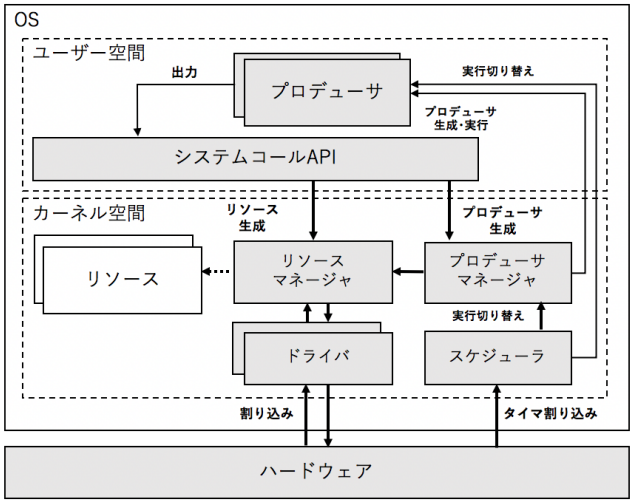

図 3 R2OS のアーキテクチャ

3.3.2 スケジューラ

スレッドの実行を管理する．スレッドとして実行されるのは, プロデューサとプロデューサマネージャである．実行はラウンドロビンスケジューリングによって並列で行い，時間単位で平等に複数スレッドを切り替える.

3.3.3 リソースマネージャ

リソースマネージャは，リソースの入出力の生成と削除を管理する．リソースの入出力を行う場合は, リソースマネージャが提供する API によって行う．またプロデューサが利用するために，システムコール API としても提供されている.

3.3.4 プロデューサマネージャ

プロデューサマネージャは，全てのプロデューサの実行の開始や終了を管理する．プロデューサマネージャ自体はスケジューラによって定期的に実行される．プロデューサマネージャが実行されると，リソースマネージャからリストに存在する利用可能なリソースの入力を受け取り，プロデューサを実行させる.

3.4 全体の動作

図 3 で表すアーキテクチャの全体の動作を 2.1 節で述べたデータ検出のアプリケーションの動作と交えて説明する．データ検出アプリケーションの 3 つのロジックの 1 つであるルーティングテーブル生成ロジックの実行を想定する．そしてユーザプログラムを持つプロデューサクラスは，既に R2OS 内部に存在しいつでも実行可能な状態であるとする．この時アーキテクチャ全体の動作は図 4, 5, 6, 7 の, ハードウェアの入力, プロデューサの生成, プロデューサの実行, ハードウェアへ出力の 4 つに分けられる．まずアプリケーションの実行は，シンクノードのブロードキャストによって開始する．ノードがブロードキャストのデータを受信すると，図 4 で示す通りハードウェアの割り込みによってネットワークリソースのドライバが実行される．ド

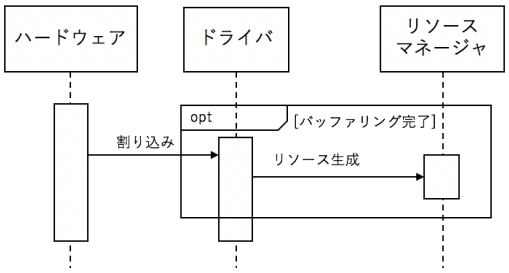

図 4 ハードウェアの入力のシーケンス図

ライバは，受信したデータを内部でバッファリングしハードウェアの仕様に従いパケットとして解析して，アドレスやパケットサイズなどを含むネットワーク情報として抽象化をする．そしてドライバから，リソースマネージャの API を利用して出力のリソースを生成し，そのリソースにネットワーク情報を書き込む．リソースマネージャでは，生成されたリソースは利用可能状態となってリストに格納される．次にタイマ割り込みによってスケジューラが呼ばれる．スケジューラは定期的に実行され，プロデューサとプロデューサマネージャを切り替えて実行する．そこでまずは図 5 で示すプロデューサマネージャが実行されるとする．プロデューサマネージャは，リソースマネージャからリストに存在する利用可能状態のリソースを受け取る．リソースがプロデューサの入力として全て存在した時，そのプロデューサを生成する．この時プロデューサはスレッドとして実行するため，スケジューラにあるスレッドのリストに保存される．図 6 で示す通り再度タイマ割り込みによってスケジューラが実行されると，スレッドのリストにあるプロデューサの実行を開始する, または既に開始されている場合は実行を再開する．プロデューサはネットワークリソースとメモリリソースの 2 つに出力するため，システムコール API を通じてリソースマネージャから出力リソースが生成される．この時のリソースはまだ利用可能状態でなく，プロデューサは自由にデータを書き換えることができる．プロデューサの実行が終了すると，生成された 2 つのリソースは利用可能状態となりリソースマネージャがドライバに出力する．また終了時には, リソースマネージャにリソースの削除依頼を出し, 入力リソースが他のプロデューサに使われていない場合はメモリから削除する．最後に図 7 で示す通り, 割り込みによって実行したドライバは出力として受け取ったリソースを利用して効率よくハードウェアに書き込む．書き込みが終了した時に出力が他にプロデューサが利用されていない場合は, 出力をメモリから削除する．これによりネットワークリソースとメモリリソースを入出力とするプロデューサが R2OS で動作する．

4. 利用手順

アプリケーション開発者が R2OS を実際に利用する場合は, ユーザプログラムであるプロデューサの用意とプロデューサは何を入出力とするかを R2OS に知らせる必要がある．まずプロデューサを登録する手順について説明する．次のプログラムはプロデューサを登録する擬似プログラムである．

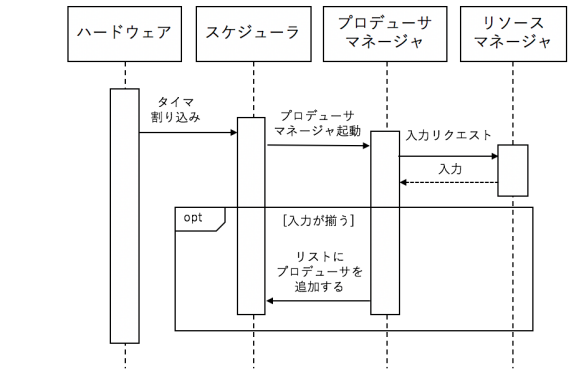

図 5 プロデューサの生成のシーケンス図

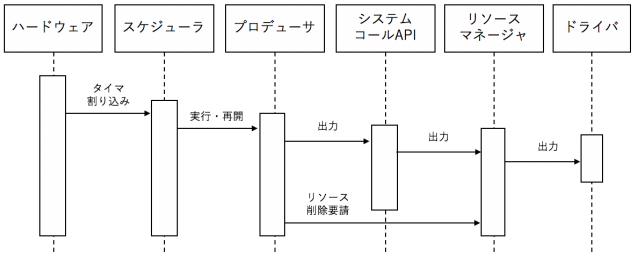

図 6 プロデューサの実行のシーケンス図

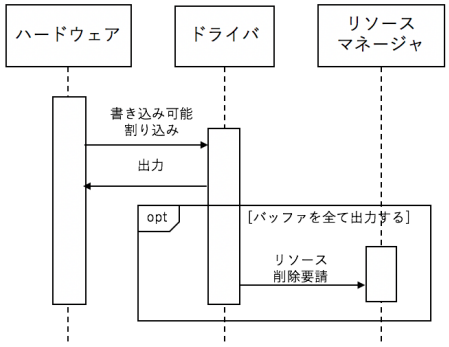

図 7 ハードウェアへの出力のシーケンス図

```
1  resource_t in[] = {
2    RESOURCE_MEMORY,  // 入力(整数値)
3    RESOURCE_SENSOR,  //
4  };
5
6  resource_t out[] = {
7    RESOURCE_NETWORK, // 出力(整数値)
8  };
9
10 attach_producer(
11   in, GET_ARRAY_LENGTH(in),
12   out, GET_ARRAY_LENGTH(out),
13   transfer); // ユーザプログラムポインタ
```

プロデューサは, 10 行目の R2OS のシステムコールである attach_producer を呼ぶことでを登録できる. プロデューサの追加には, プロデューサのリソースの入力と出力を指定する. この入出力はリソースの種類を表し, 擬似プログラムでは, 2 行目のメモリリソースからの入力と, 3 行目のセンサリソースからの入力によって実行することを宣言している. そして, 13 行目で示す通りメモリに展開されたプログラムのアドレスを引数に渡す. これは Flash メモリまたは RAM に展開されたプログラムがプロデューサとして実行されることを表している. 次に実際のプロデューサプログラムをどの様に書いて実行されるかを説明する. 次の疑似プログラムは, 2.3 節で述べたプロデューサの 1 つ, センサーリソースからデータを検出してネットワークリソースに送信するプロデューサである.

```
1  void transfer(producer_t *context){
2    // センサリソースとメモリリソースの入力
3    resource_t *in_m = context->inputs[0];
4    resource_t *in_s = context->inputs[1];
5
6    // 入力のデータ
7    table_t *table =(table_t *)in_m->bytes;
8    sensor_t *sensor = (sensor_t *)in_s->bytes;
9
10   if(routing_table->size == 0){
11     return;
12   }
13
14   // 出力用のバッファを用意
15   resource_t *out = create_resource(
16     context,
17     0,
18     sizeof(req_t) + sizeof(sensor_t));
19   req_t *req = (req_t *) out->bytes;
20
21   // バッファにデータを出力
22   memcpy(
23     table->sorted_nodes[0].addr,
24     req->addr,
25     sizeof(req->addr));
26   memcpy(
27     sensor,
28     out->bytes + sizeof(req_t),
29     sizeof(sensor_t));
30 }
```

メモリリソースとネットワークリソースにアクセスするには, 1 行目の引数で渡されたコンテキストを利用し, 3, 4 行目のように参照することできる. inputs は配列であり, プロデューサ登録時の入力の順番で格納されている. 次にそれぞれの入力をキャストすることでデータの構造にアクセスできる. ネットワークリソースの入力のデータ構造はリソースの仕様で決まる. そして出力は, 15 行目で示す通り R2OS の提供するシステムコールを用いて出力用のバッファ生成し, そのバッファにデータを書き込むことで行う. バッファを用意する場合は, バッファのサイズとプロデューサ登録時の出力のインデックスを指定する. プロデューサの出力はネットワークリソース 1 つとして登録してあるため, 17 行目で示す通りインデックスの値は 0 となる.

表 1　Arduino Mega の性能

CPU	SRAM	Flash Memory
atmega2560	8KB	256KB の内 8KB がブートローダ

5. 実装

本研究では Arduino Mega[9] を利用して R2OS を実装した. 表 1 は Arduino Mega の仕様である. 本節では R2OS の実装による仕様について説明する.

5.1 固定リソース・ドライバ

ドライバは，固定のリソースとしてネットワークリソースと，インメモリストアを抽象化したメモリリソースの抽象化を実装した. リソースの種類は整数値であり，実装した各リソースの割り当てはネットワークリソースとメモリリソースのそれぞれを 1，11 とした. ネットワークリソースは，Arduino Mega では USART ポートをサポートしているためシリアル通信を抽象化したものである. インメモリストアは，RAM 領域で 128 バイトを確保しドライバで抽象化したものである.

5.2 プロデューサマネージャ

プロデューサマネージャは，リソースからの入力とプロデューサを管理する. 入力リソースは, リソースの種類, データ, 状態, そして参照カウンタを持ち, 最新のものを末尾としたリスト構造になっている. 入力リソースの状態は, 準備中, 利用可能, 利用中の 3 つ存在し, プロデューサマネージャが状態を変化させる. ドライバによって入力リソースが生成された場合は, 状態は利用可能となる. 利用可能の状態の場合は, プロデューサはリソースを入力とすることができ, プロデューサマネージャはプロデューサを実行する. プロデューサが出力によって別のプロデューサにとっての入力リソースが生成された場合では, 状態は準備中でありプロデューサはリソースを入力にできず, プロデューサが終了した段階で, プロデューサマネージャが利用可能状態にする. 参照カウンタは入力リソースを利用しているプロデューサの数を表し, ライフサイクルと関係している. 入力リソースは複数のプロデューサが参照するため, 全ての参照しているプロデューサが実行を終了すると削除される. 入力リソースを利用する上で 1 つ問題が存在する. 入力リソースは, メモリを確保するため同時に生成される最大数が決まっている. しかし, 複数の入力を持つプロデューサの 1 つの入力が, 短い間隔で定期的に実行されるとき, 入力が大量にバッファリングされることでメモリが枯渇する可能性がある. そしてそういった定期的に実

行されるリソースでは, 入力が最新のものだけで良い場合が存在する. 例えばタイマ割り込みは, タイマ割り込みが発生する度に入力リソースを生成するのではなく, 最新の入力リソースのみを残し, それまでの入力リソースは破棄する方が都合が良い場合がある. こういった状況を想定し, 入力リソースの生成時にユニーク制約をつけることができる. ユニーク制約がつけられた入力リソースは, リソースの種類が同じ既に存在する入力リソースに上書きされる. しかし, 既に入力リソースがプロデューサによって利用されている場合は, 入力データが一貫性を保てなくなるため, 入力リソースの状態から判断して, 既に使われている場合は新しい入力リソースを生成する.

6. 評価

本節では R2OS を実装のイメージサイズ，実行時のメモリの使用量，そしてアプリケーション開発における利点で評価する．まず R2OS のイメージのサイズは 77,368 バイトであった．Arduino Mega では表 1 で示す通り Flash メモリが約 256 キロバイトであるため，R2OS は利用可能容量の 1/3 以下である．これは現実的なイメージサイズであり，更にドライバを追加してしても問題ないと考えられる．

次に実行時のメモリ使用量について評価する．R2OS では静的なデータと動的なデータを確保していて，静的なデータのサイズはスレッド，リソース，プロデューサ，ドライバに関係し，ドライバを除きそれぞれ最大数を定めているため実行以前から算出できる．ドライバで利用する静的なデータは, ハードウェア構成によってメモリの確保量が変化するため, ここでは考慮しない．スレッド数は最大 8 つまで同時に実行でき，1 つのスレッドによる静的なメモリ使用量は 143 バイトであるため，合計で 1144 バイト使用する．リソースは，1 つのリソースの入力のヘッダが 9 バイトで最大数は 64 であるため，合計で 576 バイト使用する．プロデューサは実装上ではプロデューサクラスとプロデューサインスタンス別れていて, それぞれ 1 つに対して 20 バイトと 18 バイト使用していて，最大数はそれぞれ 6 と 7 であり，合計で 246 バイト使用する．その他にリソースやプロデューサのアルゴリズムのために利用するグローバル変数等を含むと，全ての静的なデータの合計は 1976 バイトとなる．Arduino Mega の SRAM の容量は 8 キロバイトであるため，SRAM の約 25 %のメモリを利用する．そして残りの利用可能なサイズは 6024 バイトとなる．動的なデータの確保では, プログラム実行時のスタック利用量を図るのは困難であるため, ヒープ領域のみ使用を評価する．スタック領域やドライバ内のメモリ確保は大きく見積もり最大 2048 バイト利用されるとすると, ヒープ領域で利用できるサイズは最終的に 3976 バイトとなる．ヒープ領域の利用は, リソースの入力データとプロデューサのプログラムデータで使用する．リソースとプロデューサの実行は最大数利用されているとすると, リソースのデータとプロデューサのプログラムデータの利用可能なデータサイズの関係は図 8 となる．X 軸はリソースの確保するサイズの平均値で, Y 軸はその時のプロデューサプログラムの利用可能なサイズの平均値である．中間値としては, リソースは約 30 バイト, プロデューサプログラムは約 300 バイト利用可能であり, 1 のプロデューサに対して単純なロジックのプログラムであれば, メモリは足りると考えられる．またこれはリソースとプロデューサを最大数利用した場合であり, これ以上に多くのメモリの使用を期待できる．

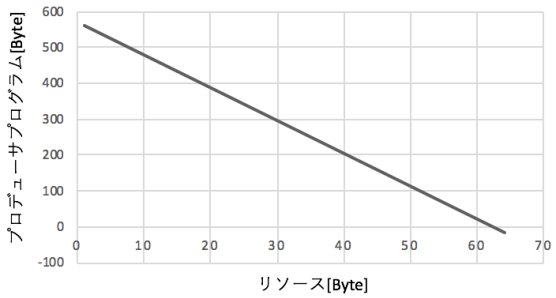

図 8 リソースとプロデューサのプログラムの利用可能なサイズの関係

最後に R2OS を利用する利点について評価する．R2OS は一般的に開発するよりも再利用性・品質管理・保守性の点で優れている．まず再利用性において，プロデューサのユーザプログラムは参照透過が常に保たれるため，入出力のインターフェイスを守ることでロジック部分の置き換えが容易である．2.3 節の例では，ルーティングテーブルの生成ロジックをより効率的なプログラムへ置き換えた場合でも，アプリケーション全体を変更することなく再利用可能である．次に品質管理においても，プロデューサの参照透過性が有効である．R2OS はハードウェアをリソースとしてアプリケーションと明確に分離されているため，実環境を模したシミュレータ上でのテストが容易で，また実行に副作用を含まないため不具合の発生を再現しやすく，デバッグしやすいことが挙げられる．最後に保守性においては，アプリケーションをランタイムで更新可能な点が挙げられる．近年のデバイスの性能向上から WSN の要件の 1 つとして再プログラミング可能な点が存在する．WSN の性質上，大量のデバイスを投じた後にソフトウェアの不具合が生じた場合は，デバイス上で動くプログラムをネットワークを通じてランタイムで更新する必要があり，複雑なロジックになりやすい．R2OS ではハードウェアなどへのデータの書き込みは，OS 上のドライバにより安全に行われるため，プロデューサによる出力は結果整合性が保たれる．そのためプロデューサがランタイムで更新する場合でも，複雑なロジックを考慮することなく中断できデータの損失を防ぐ．また R2OS の実行モデルから必然的にプロデューサのロジックの粒度が低くなることから，アプリケーショ

ⓒ 2018 Information Processing Society of Japan

ン全体を更新することなく，必要箇所のプロデューサ単位での更新が可能となり，ネットワークリソースのコスト削減に繋がる．

7. 関連研究

既存の研究で提案されている WSN の OS に, Contiki と TinyOS[10] が存在する．両方の OS は共にプリエンプティブスケジューリングによるマルチスレッドを実現している．TinyOS はハードウェアをコンポーネントとして, コンポーネント同士を接続することでアプリケーションを実現しており, R2OS の実行モデルと似ている．また TinyOS の拡張である TOSThreads では, ラウンドロビンスケジューリングも可能としている．しかし, 並列で実行されるプログラムが利用する共有リソースの管理の仕方が異なる．Contiki と TinyOS は共有リソースの管理をアプリケーション側に任せており, ユーザプログラム上でミューテックスやセマフォを利用して管理する必要がある．R2OS では OS が共有リソースを管理しているため, アプリケーション開発においてその事を考慮する必要がない．

8. まとめ・今後の課題

近年 IoT の発展に伴い WSN についても再び注目され，様々な要件で利用されてきている．しかし WSN のデバイスは，大量のデバイスを投じるため最小限の安価なハードウェアモジュールで構成されるため，複雑な要件を実現するためにはハードウェアの直接的な制御や最適化を行う必要があり，アプリケーションの開発コストを高めている．そこで既存の研究では，WSN 向けの OS を提案しており，ハードウェアを抽象的に扱える一方，一般的な OS よりもオーバヘッドの少ない実装を実現している．一般的に WSN 向けの OS はプリエンプティブスケジューリングをするように実装されており，複雑なアプリケーションや大規模開発になった場合には複数のスレッド間でのリソースの共有管理が困難となる．また WSN の要件である再プログラミングにおいて，アプリケーションの全ての実行を中断し，プログラム全体を更新する必要があり，データロスや通信による大幅なコストが発生する．そこで本研究では，WSN 向けのプロセスによる新しい実行モデルとその OS である R2OS を提案した．R2OS は，ハードウェアを入出力を持つリソースとして抽象化し，ユーザプログラムであるプロデューサをリソースへの入出力をつなぐパイプラインとしてプロセス実行する実行モデルである．これにより，ロックといった共有リソースを管理をする必要がなく，また再利用性・品質管理・保守性に優れている．

今後の課題では，R2OS ではスレッド実行する場合に，1 つのスレッドに対するスタックサイズが固定化されているため，過剰に領域を確保されてしまう問題を解決する必要がある．また本研究の実装では，1 プロデューサあたりの明確なメモリー空間の分離はされているが，アプリケーションプログラム実行で破壊的な書き込みが行われる可能性がある．そのためプロデューサ上で実行される仮想マシンが必要である．その他にプロデューサの実行による出力は結果整合性のみを保証しているため，アプリケーションの要件によっては明示的もしくは暗示的なロックの仕組みとトランザクションのサポートを行う必要がある．

参考文献

[1] Lazarescu, M. T.: Design of a WSN Platform for Long-Term Environmental Monitoring for IoT Applications, *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, Vol. 3, No. 1, pp. 45–54 (online), DOI: 10.1109/JETCAS.2013.2243032 (2013).

[2] Javed, F., Afzal, M. K., Sharif, M. and Kim, B.: Internet of Things (IoT) Operating Systems Support, Networking Technologies, Applications, and Challenges: A Comparative Review, *IEEE Communications Surveys Tutorials*, Vol. 20, No. 3, pp. 2062–2100 (online), DOI: 10.1109/COMST.2018.2817685 (2018).

[3] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. and Pister, K.: System Architecture Directions for Networked Sensors, *SIGARCH Comput. Archit. News*, Vol. 28, No. 5, pp. 93–104 (online), DOI: 10.1145/378995.379006 (2000).

[4] Buttazzo, G. C., Bertogna, M. and Yao, G.: Limited Preemptive Scheduling for Real-Time Systems. A Survey, *IEEE Transactions on Industrial Informatics*, Vol. 9, No. 1, pp. 3–15 (online), DOI: 10.1109/TII.2012.2188805 (2013).

[5] Dunkels, A., Gronvall, B. and Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors, *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462 (online), DOI: 10.1109/LCN.2004.38 (2004).

[6] Qureshi, I.: CPU Scheduling Algorithms: A Survey, Vol. 5, pp. 1968–1973 (2014).

[7] Baghyalakshmi, D., Ebenezer, J. and Satyamurty, S. A. V.: WSN based temperature monitoring for High Performance Computing cluster, *2011 International Conference on Recent Trends in Information Technology (ICRTIT)*, pp. 1105–1110 (online), DOI: 10.1109/ICRTIT.2011.5972379 (2011).

[8] Potdar, V., Sharif, A. and Chang, E.: Wireless Sensor Networks: A Survey, *2009 International Conference on Advanced Information Networking and Applications Workshops*, pp. 636–641 (online), DOI: 10.1109/WAINA.2009.192 (2009).

[9] Arduino: Arduino Mega 2560 Rev3 (2018).

[10] Levis, P. A.: TinyOS: An Open Operating System for Wireless Sensor Networks (Invited Seminar), *7th International Conference on Mobile Data Management (MDM'06)*, pp. 63–63 (online), DOI: 10.1109/MDM.2006.151 (2006).

ⓒ 2018 Information Processing Society of Japan